Epistemic Communities, Situated Learning and Open Source Software Development

Edwards, Kasper

Published in:
Proceedings from the conference on Epistemic Cultures and the Practice of Interdisciplinarity

Publication date:
2001

Document Version
Early version, also known as pre-print

Epistemic Communities, Situated Learning and Open Source Software Development

By Kasper Edwards*

Department of Manufacturing Engineering and Management, Technical University of Denmark
Building 423, office 028, 2800 Lyngby, Denmark. (email: ke@ipl.dtu.dk)

ABSTRACT

This paper analyses open source software (OSS) development as an epistemic community where each individual project is perceived as a single epistemic community. OSS development is a learning process where the involved parties contribute to, and learn from the community. It is discovered that theory of epistemic communities does indeed contribute to the understanding of open source software development. But, the important learning process of open source software development is not readily explained.

The paper then introduces situated learning and legitimate peripheral participation as theoretical perspectives. This allows the learning process to be part of the activities in the epistemic community. The combination of situated learning and epistemic communities is shown to be fruitful and capable of explaining some of the empirical observations. In particular the combination of theories can shed light on the motivational issues and group dynamics.

Keywords: Open source software, epistemic communities

Note: This is a working paper. An early draft of this paper was initially prepared for the 'Epistemic Cultures and the Practice of Interdisciplinarity' Workshop at NTNU, Trondheim, June 11-12 2001.

Kasper Edwards is a Ph.d. student at Department of Manufacturing Engineering and Management at the Technical University of Denmark. The title of his project is "Technological Innovation in Software Industry" the project is focusing on the Open Source Software development process. Kasper Edwards has a background as a M.Sc. (ENG) and has worked for several years as an independent computer consultant. He is an experienced user and administrator of both Linux and the different flavours of Win9x and NT/2000. As a hobby he maintains a PC gaming network in a youth club powered by Linux servers. As part of the project and hobby Kasper Edwards is engaged in the SSLUG (Skåne Sjælland Linux User Group, a 5000+ member user group) and participates in mostly technical discussions.

* I would like to thank associate professor Jørgen Lindgaard Pedersen for helpful comments and suggestions on earlier drafts on this paper and Ed Steinmüeller and Jane Millar of SPRU for mentioning Epistemic Communities and Situated Learning. I assume full responsibility for any remaining vulnerabilities.

1 Introduction

Computers and software are inseparable and computers are useless if there is no software to use on the computer. The term ‘open source software’ describes software with a license, which allows anyone to modify and redistribute the source code. However, open source software is better understood if one looks beyond the source code and the license, which defines the code. What we are trying to grasp is rather a phenomenon – the open source software phenomenon. The open source software phenomenon is more than just the open source software. It is also a community that have created the software and the emerging economy, which is based on open source software. Together these elements form a complex evolving system, which is intriguing and difficult to understand.

The problems of understanding the open software phenomenon are to some extent founded in the fact that open source software is free of charge. Free in this context means that the generally motivating profit is hard to spot. Software development requires resources in order to be developed. And, with little or no direct monetary reward, how is open source software being develop?

To understand this or at least to provide an initial answer we must broaden the perspective to the level of the phenomenon. Part of the open source software phenomenon is the open source software community or more precisely large number of small communities, each developing their own open source software to work with the software of the other communities.

There have been several attempts to present a theory for understanding of open source software development. Just to mention a few, some use non-conventional economic (Raymond, 1999a) theory or theories from psychology (The Linux Study), anthropology (Raymond, 1999b), the economics of to career concern (Learner and Tirole, 2000), leadership

(Edwards, 2000a), actor network theory (Tuomi, 2000) and the free provision of public goods (Bessen, 2001 and Edwards, 2001).

These different approaches capture different perspectives, which explain different parts of the reasons for the existence and continued development of open source software. The epistemic communities' approach applied in this paper is yet another way of understanding the open source phenomenon.

In this paper, I propose that part of the understanding of the open source software phenomenon should be found in the communities, which develop open source software. I examine the open source software development process as a community process. It is shown that open source software communities closely resemble epistemic communities. Analysis of open source software development as an epistemic community provides insight into how it is possible to develop open source software, and does to some extent shed light on motivational issues as well.

However, there are shortcomings in the epistemic communities' approach. It is difficult to explain the entry of new developers into the epistemic community and as a consequence hereof legitimate peripheral participation and situated learning are introduced. This extends the usability of the theoretical framework and provides further insights into the epistemic communities' approach to understanding open source software development.

This paper begins by introducing open source software and by explaining how it differs from commercial software. The common development methodology of open source software is then described. Next, the understanding of epistemic communities is presented and open source software is analysed. Situated learning is incorporated at a later stage, and it is explained how this contributes to the understanding of the open source phenomenon.

2 Introducing Open Source Software

Open source software is software like all other software: A sequence of instructions to be interpreted by a computer, which performs actions accordingly. Software exists in all shapes and sizes and is made to solve different tasks. No piece of software exists that covers all areas of use, and still more software is being developed as computer are becoming increasingly more important in our daily lives. Most software solves a specific purpose. Like word processing software solves the problem of writing text, creating layout for the text, email clients are software for sending and receiving emails, and the list goes on. Software helps people perform different tasks, some of which can only be performed by use of a computer with the proper software. Other tasks are made easier and faster resulting in a productivity gain. Given the increasing number of computers in society, it is no wonder that software is perceived as a good, which is often both attractive and necessary.

Open Source Software (OSS) is computer software, which comes with a license that is very different from the licenses in commercial software like Microsoft's Office suite or the Windows™ operating system. The license is a legal agreement between the user and the producer. The license defines the terms of use, which a user must accept to be allowed to use the software. Commercial companies have typically relied on very strict licence schemes, which allowed the user a minimum of rights. Commercial licenses do not allow users to copy or modify the software in any way. Users are only allowed to use the software. The economic rationale is understandable, as copying would reduce sale, and modifications imply some kind of reverse engineering, which reduces the competitive advantage of the software selling company. Commercial companies never release the source code for their software. The source code for software is the human-readable instructions, which make up the software, before it is translated into something computer-readable (a binary file). Source in hand, a skilled person will be able to understand how the software has been constructed and make

desired modifications. Companies treat source code as business secrets, as all software related development efforts are reflected in the source code. The source code for software may be compared to the blueprint of a construction. In order to use software source code has to be compiled. When source code is compiled, a special program (a compiler) translates the human-readable source code into a machine-readable code, which a computer can understand.

Historically open source software has existed since the early days of computing as software sharing. In the early days of programming at MIT students and members of the modelling railway club (Levy, 1984) shared their software. There was a common understanding of learning from each other and elaborating on each other's software. In the early 1970'ies when the Unix operating system was born at AT&T, universities were allowed to use it free of charge. The source code was free and the universities contributed back to AT&T (for an in-depth description of the early days of Unix (Salus, 1994 and Edwards 2000b)

The name "open source software" hints one of the special properties of the license: The source code for the software is freely available – open source. Apart from access to the source code, open source software licenses have other properties, which are very different from that of the usual commercial software license. An open source software license grants the user:

1. The right to free redistribution
2. Access to source code, which may be modified
3. The right to create derived works
4. The obligation to redistribute the license

This is an adapted excerpt from the open source definition[1] as published by the Open Source Software Initiative showing the most important features. An open source software license is clearly very different from its commercial counterparts. None of the mentioned four properties are ever granted in commercial software licenses. The right to free redistribution means that any person, who accepts the license, is allowed to make as many copies as he wishes and distribute these copies at any price he may choose. Access to source code is required, and any person creating open source software has to make the source code for his software available to anyone. The right to create derived work allows anyone to use the source code from open source software, modify the source code and distribute this work under a new name. It is, however, required that reference the original contributor are made. The original contributor retains copyright for the code, which he wrote. To keep the software from changing license to a non-open source license it is required that the license is distributed with the software. A person making copies of open source software is therefore required to copy the license.

Essentially open source software relies on basic copyright law to grant users the above-mentioned rights. The creator of any software is by law granted copyright for any code that he might produce. Copyright grants the creator exclusive rights to produce copies of his creation and make his creation available to the public. Thus no one is allowed to make copies or distribute creations, which they do not themselves have copyright for. Open source licenses explicitly surrender some of these rights to the user, which are allowed to make copies and distribute these. The Free Software Foundation has named this copy*left* as it is opposite to copy*right*.

[1] The Open Source Definition is used to define what should be understood as Open Source Software, see, URL: http://www.opensource.org .

2.1 The Organisation of Open Source Software Development

Open source software development are organised around numerous projects, each project aims at providing a specific piece of software. There are no central authority that initiate projects or decides who does what and how. Anybody can announce a project or make available software and call it a project. In this respect open source software development are completely autonomous.

Projects serve to distinguish between different software and different efforts to archive a specific goal. As such projects differ greatly in size and scope for instance, the GNU Project[2] aims at developing a complete Unix-like operating system, which is free software. The halfd[3] project is much more narrow in its perspective and aims at providing a server management tool for Linux half-life servers. Projects then seem like a basic unit of understanding when analysing the open source software phenomenon.

2.2 The Open Source Software Development Model

Open source software exists in all sorts and sizes, and every day new software projects are initiated. Most of these projects are undertaken by people doing it free as a kind of hobby. It is important to emphasise that the nature of the activities in OSS projects is a development effort. It is the objective of open source software projects to create some particular software. When the software has been developed, the projects tend to die a peaceful death where participators gradually loose interest and turn away from the project. This by no means indicates that the resulting software is no longer being used, rather that the software has reached a state of maturity (Edwards, 2001, pp.40-43), where most users are satisfied and have no need for further development.

---

[2] The GNU project, www.gnu.org

[3] The halfd project, www.halfd.org

A common feature of OSS development projects is the means of communications employed. Communications in OSS development are maintained primarily using services facilitated by the Internet such as email, newsgroups, mailing lists, web pages, and chat. Of the mentioned forms of communication, chat is the only real-time medium. Occasionally telephone is being used, but this is a rare event mostly due to the cost of long distance phone calls and the intimate nature of personal phone calls. FreeBSD developer Poul-Henning Kamp (2000) noted that phone calls were invaluable when trying to solve a complex problem, but did require a level of personal intimacy not usually associated with OSS development. Phone calls also moved the development process out of the regular forum and isolated the discussion from the project.

Most projects use a web page for general information and download of the project software. Centralised means of communications are facilitated by mailing lists, which provide a centralised way of reaching all the people in the project. Email sent to the mailing list address is automatically resent to everyone who has subscribed to the mailing list.

Within open source software projects there can be identified three general modes of organisation: 1) Free for all, 2) Core team, 3) Maintainer based. The organisation reflects who and how decisions are made about the project development i.e. what code goes into the project and what should be left out. Free for all projects rely on an automated system for keeping track of changes in the project code. The other types of organisation could also employ these concurrent versioning systems (CVS) but they are imperative in free for all projects are there are no person to filter incoming code. The important distinguishing feature in free for all projects are that anyone have equal permissions to add code to the project. Core team based projects place restrictions on the general public and rely on a core team to decide what code is allowed into the project. Core team based projects could use CVS and would

then only allow the system to accept code from the core team members. The last general form of organisation is the maintainer-based projects, which will be discussed in this paper.

Most open source software projects are organised with a central person regarded as responsible for the project. This person is referred to as the maintainer. The maintainer is often the person who conceived the idea for the software, which the project is trying to create. It happens that a maintainer grows tired of a project and passes on responsibility to another person who then becomes maintainer. The maintainer takes on a special responsibility for the project and functions as the personal point of contact for the project. The maintainer is the person who releases new and improved versions of the software. As such the maintainer has the final word on what features and suggestions should be incorporated into the software. People, who participate in the project, are referred to as contributors. Contributors send improvements and added features in the shape of a patch to the maintainer. A patch is a file containing source code, which is intended to be merged with the project source code. The basic development cycle in open source software development is as follows:

1. Maintainer releases software and source code
2. User downloads software and source code
3. User identifies problems or needed features
4. User implements changes
5. Contributor returns changes to the project for inclusion
6. Corrections are discussed on the mailing list
7. Maintainer reviews the corrections and includes changes in the project software
1. Maintainer releases new version software and source code

2. User downloads software and …… (and so forth)

The maintainer initiates the projects by making available the first version of the software. Usually this implies that some sort of project homepage has been set up and that announcements have been made on various news groups, mailing lists and homepages. Interested users download the software to try it. Some users immediately dislikes the software and deletes it, others find it useful and begin to use the software. Some of those who find the software useful also identify problems or needed features in the software. Interested users then subscribe to the project mailing list to receive discussions regarding the project and its further development. Blessed with the ability to write code and access to the source code the interested user now begins to make changes to the software. Once changes are made, the contributor has a choice whether to contribute to the project or retain the changes to him self. If the user decides to submit the changes (referred to as a patch) to the project, the user is then becomes a contributor to the project. Usually patches are emailed to the project mailing list or the project maintainer. It is the hope of the contributor that his patch will be included in the project. The maintainer and/or other persons on the mailing list will review the submitted patch and discuss it. The interest in the submitted patch is co-related to the interest, which other people have in that particular area of the code where the changes have been made. One some occasions the code will not be discussed in detail, simply because no one on the list is interested in that part of the software. Usually the maintainer is interested – it is after all 'his' project – and the maintainer will make sure that the software does not develop in an undesired direction. Following review and perhaps some discussion, the patch is either accepted or rejected. If accepted the patch is included in the software, and the new version is made available for downloading.

Often several or even hundreds of persons are engaged in a development project and contribute to the best of their efforts. Large variations in contributed code can be observed in

the projects. In the GNOME[4] project contributions from each developer have been analysed, and there is clear evidence of an inner circle of programmers responsible for a proportionally very large part of the code (Koch, & Schnider, 2000). People who contribute code to the project also participate in discussions on the mailing lists. Contrary to what one might think, there is no co-relation between the lines of code contributed by a developer and his activity on the mailing lists (Koch, & Schnider, 2000).

Participation in development requires a person to follow the development and discussions regarding the project. Projects with many active developers often show a very high volume of emails going to the project mailing lists. The Linux kernel development is an example of a project with a high volume mailing list. In week 20 year 2001 the Linux kernel mailing list received 1227 posts from 423 different contributors 45% posted more than once, and 38% posted in the week before (Kernel Traffic, 2001). This suggests that many regulars frequent the list. The sheer amount of emails to the list is enough to discourage many people; receiving approx. 1200 emails a week from one list alone seems overwhelming. Despite moving specific discussions to other mailing lists and creating separate projects when appropriate, the kernel mailing list continues to have high volume. The number of posts (emails) in week 20 (Kernel Traffic, 2001) is on average, although recent weeks have seen up to 1750 posts. However, most projects are rather small and following the project development does not require prohibiting amounts of time.

[4] The GNOME project is trying to create a common graphical desktop for Linux. See, URL: http://www.gnome.org

3 Epistemic Communities

The term epistemic communities originate in political science and have been used to understand how people with knowledge in a particular area and a common interest form communities and influence policy makers.

The understanding of epistemic communities in this paper is highly inspired by the works of Haas (1992) and Holzner & Marx (1979). An epistemic community may consist of participants from various disciplines with various previous experiences, and they have the following four characteristics (adopted from Hass, 1992):

1) A shared set of normative and principled beliefs, providing a value based rationale for the social action of community members;
2) Shared causal beliefs, which are derived from their analysis of practices leading or contributing to a central set of problems in their domain and serving as the basis for elucidating the multiple linkages between possible policy actions and desired outcomes;
3) Shared notions of validity – that is, intersubjective, internally defined criteria for weighing and validating knowledge in the domain of their expertise; and
4) A common policy enterprise – that is, a set of common practices associated with a set of problems to which their competence is directed, presumably out of the conviction that human welfare will be enhanced as a consequence.

The major dynamics are uncertainty, interpretation, and institutionalisation, which determine the creation of epistemic communities. In the framework of Haas, epistemic communities are largely motivated by the existence of a power vacuum that has come about in the face of uncertainty. In this situation policy makers turn to epistemic communities for information (Haas, 1992, p.15) to ameliorate uncertainty. When satisfying the need for information, epistemic communities emerge, proliferate (Haas, 1992, p.4), and prevailing epistemic communities become strong actors in decision making. In Haas (1992, p.4) words:

> "Epistemic communities are one possible provider of this sort of information and advice. As demand for such information arise, networks or communities of specialists capable of producing and providing the information emerge and proliferate. The member of a prevailing community become strong actors at the national and transnational level as decision makers solicit their information and delegates responsibility to them."

When facing problems with complex interlinkages policymakers can use epistemic communities to help interpret and formulate alternative causes of action. If there is a continued need for information and interpretation epistemic communities tend to become institutionalised.

The motivating factor in the emergence of epistemic communities is influence on policy making, being able to influence policy is in itself a reward. With influence follows recognition from ones peers, and access to funding helps the emergence of a community sharing the same ideas. There are obvious elements of a reputation game between actors trying to archive influence, recognition, and reputation amongst ones peers. Scientific communities are indeed epistemic communities, where actors are motivated by influence and reputation.

In open source software development, the motivating factor is also influence, recognition from ones peers, and reputation. Eric S. Raymond (2000) is very explicit in relating motivation in OSS development to a reputation game in the gift economy:

> "There are reasons general to every gift culture why peer repute (prestige) is worth playing for:
>
> First and most obviously, good reputation among one's peers is a primary reward…..
>
> Second, prestige is a good way (and in a pure gift economy, the *only* way) to attract attention and cooperation from others. If one is well known for generosity, intelligence and fail dealing, leadership ability, or other qualities it becomes much easier to persuade other people that they will gain by association with you"

Unlike Haas epistemic communities, influence is directed at the software project, that is, the community and the software in it self and not policy making. At present, the open source software community does not offer relevant information for policymakers. This situation may soon change as a recent report to the European Parliament on the ECHELON Interception System concludes (Temporary Committee on the ECHELON Interception System, 2001):

> "The Commission and Member States are urged to devise appropriate measures to promote, develop and manufacture European encryption technology and software

> and above all to support projects aimed at developing user-friendly ***open-source encryption software***." (Emphasis added)

When promoting open source encryption software the open source community may offer relevant information to policymakers. However, the area of encryption software is very small and may only be able support a very small number of open source software developers. At present, it must therefore be concluded that OSS development does not attract enough attention from policymakers to motivate the creation of communities as described by Haas. Consequently, the political establishment does not offer any monetary incentives for open source software development in general.

Apart from receiving monetary support from the political establishment, there is the possibility of selling the project software. Open source software developers have little hope of making monetary profits from developing software. Developers are free to sell their open source software at whatever price they feel right, but the source code is freely available, and everyone is allowed to make copies for free. This makes it difficult to actually sell the software as potential buyers will be able to get the source code for free and the incentive to buy seems non-existing. Thus, efforts must be motivated by something else than profits from selling open source software. This might indeed be a much more personal need: The need to solve a particular programming problem.

I propose *personal need* as a fourth underlying dynamic, which stimulates the creation of OSS epistemic communities. Since policy makers do not at present demand services from the OSS community, personal need to solve a programming problem should then be the first mentioned underlying dynamic. A personal need for a specific functionality is the reason for creating a first version of some software, which in turn becomes an open source software project. When the project has been created the other dynamics begin to influence, however,

the influence exhibited in open source software development is a bit different from what is proposed by Haas.

In the open source software community many members are knowledgeable and able to articulate their needs, and they also have the ability to make their own software. No logic determines who makes the first version of software, which solves the personal need. Clearly, a person must feel a very strong need to go though the trouble of making a first version of the software.

When the first version is available, uncertainty begins to play an important part. When people begin to join the newly started project, uncertainty shows itself as general uncertainty about the function of the software and the direction of the project. What follows are a series of discussions about the direction of the project and how the project should proceed

This sparks a process of interpretation where a common understanding of the project emerges and the participators define their area of interest.

This discussion/interpretation takes place on two levels: 1) actual discussions and 2) code submitted to the project. In the actual discussions people voice their opinion and express their needs and wishes for the project. Discussions of course influence the direction of the project but the discussions are more a display of personality where the participators get to know each other. Open source software development is voluntary and discussions are only a show of promise - no one is obliged to deliver.

Code submitted to a project are a much more pervasive means of influence as actual code is a result of a real development effort. Compared to just discuss and talk about what should be done code shows how a proposition could be implemented. Code is also a clear statement about a developer's commitment to the project and the direction that he would like the project to go – the code speaks louder than a thousand word.

Participators are interpreting the contributions to the project and forming an understanding of the direction of the project. Also this process marks the areas of interest to the different developers. As contributions to the project are submitted and included in the project institutions emerge. Some contributors are very interested in certain areas of the code and will regard these areas of the code as their own. Therefore, questions and modifications to that area of the code should go through the particular contributor. This kind of contributors is perceived as maintainers of their particular area of code and becomes an institution within the project.

3.1 OSS projects as Epistemic Communities

OSS projects consist of people working together to create a particular piece of software. The primary motivation for people to enter open source software development is a personal need for particular software or functionality. The communication in the project is directed at solving problems and proposing new features and fixes for the project. There are two types of discussions in open source software projects, which are often separated into different mailing lists: 1) User support and 2) Development activities.

User support is mostly for newcomers to the project who are trying to use the project software, but are having problems making the software compile or run. Often this type of discussion is maintained on a separate mailing list and the most frequently asked questions are collected in a FAQ (Frequently Asked Questions). Often seasoned contributors as well as intermediate users in the project offer their advice in these forums. The lists form a community help line and are very common to open source software projects.

The mailing list for development activities is for the more seasoned contributors, which are engaged in the actual development of software. The discussions on this kind of mailing list tend to be very technical and code fragments and patches are the order of the day.

Discussions about the general direction of the project are common on this list. Some projects like the Linux kernel are very explicit about the general direction, and huge discussions tend to erupt, when proposals are made.

Members of the community can be identified by their subscription and participation in discussions on the mailing list(s). I define two types of actors: 1) Users and 2) Contributors.

Users are people who just want to use the software and do not care about its further development. These people download the software, install the software, and just use it without commenting further or providing any feedback to the project.

Contributors are people who take an interest in the project, follow the discussions on the mailing lists, and voice their opinion. Contributors need not contribute actual code to the project, they may just participate in discussions and stimulate debate within the community. Some contributors' just answer questions about problems related to general use and contribute to the community in this manner. Contributors are also persons, who post questions to the mailing lists and by doing so provide valuable information about the usability of the project software and documentation.

Although users are per definition non-vocal in the projects, they are an important resource for the project. As time goes by some users begin to take an interest in the project and voice their opinion, and thereby they become contributors. This could happen in a situation where a new version is released and a user realises that the new version of his favourite software does no longer run on his system. This situation often motivates a user to become a contributor and post error reports to the project mailing lists in order to get the problem fixed. The user may also posses programming skills in which case he may review the source code and fix the bugs (errors in the software). The user has a choice whether to fix the bugs on his personal system or to send the fixes to the project mailing list for inclusion in the project code base. The advantages of just fixing the problem on his personal system is a

short term gain, since the problem could be fixed immediately without having to interact with the project mailing lists. However, if the bug fixes are not included in the project code base, the user will have to make these fixes again next time the project releases a new version. The new version may have changed to such a degree that the fix does no longer apply and has to be modified to fit the new version. Thus, there is an incentive to get the fix included in the project code base. Needles to say, contributing code to a project also provide status and recognition among peers in the project, which is rewarding in itself.

The deeper understandings of open source software development projects arise when we analyse the four characteristics of epistemic communities as defined by Haas (1992) (shared normative and causal beliefs, notions of validity and common policy enterprise). These provide insight into the workings of open source epistemic communities. The following sections discuss the four points of Haas (1992) and explain how these characteristics are present in the community, and how they influence.

3.1.1 Shared Normative and Principled Beliefs

Open source software projects share the same characteristics as do the definition of epistemic communities (Haas, 1992, p.3). Contributors in open source software projects share a set of normative and principled beliefs, which provide a value-based rationale for contributing to the project. The contributors, as mentioned, do not have a simple economic incentive for contributing to the project. Contributors are at first motivated by a need for a particular piece of software to solve a problem at hand (email, word processor, etc.).

The open source software community in general (not just speaking for a single project) shares a strong disbelief in software patents and closed source software as mechanisms, which restrict the freedom of the users. Closed source software is perceived as barriers that hinder people's free choice and ability to improve software. It is very explicit in the

community that sharing code is positive and that contributing code to a project is a way of getting status within the community. Eric S. Raymond (1999a) describes the open source software culture as a gift culture, where status is derived from giving gifts to the community. Also there are evidence of a counter culture in respect to Microsoft who is perceived as dominating and always trying to pollute standards to exclude competitors.

This provides a value based rationale for being part of the community and it bonds the members of the community to share values and direct their actions towards areas perceived as valuable.

3.1.2 Shared Causal Beliefs

Contributors in open source software projects have a shared set of causal beliefs. This is related to the actual programming problem, which the project is trying to solve, but also to the intrinsic reward logic (i.e. reputation etc.). Contributors trying to solve a particular programming problem often have a background in solving that particular type of problems. This may be from formal education, previous experience with similar problems or other. They may also be newcomers to the particular area of programming and gaining experience by helping to develop the project software. This provides a common frame of reference in the projects when analysing problems and trying to create a solution. It creates an understanding of what to do to archive a certain goal. Software engineering is, however, not an exact science and there are several different principal solutions to the same problem. Often different solutions to the same problem are discussed. This gives a sense of Darwinian selection among available solutions.

The shared set of causal beliefs actually has two frames of reference, where one is the just mentioned schooling and experience. The other frame of reference is the programming project itself. The project has or will get a history of solved problems that serves as the

practise and the way to solve that kind of programming problem. Projects tend to develop a particular programming style of how programs and patches are written. This also extends to solving new problems, where previous experience in the project is a common reference and thus easily adopted.

The shared causal understanding of reward structures in the project consequently provide an understanding of other people wanting to participate in the project. It is a kind of “If I begin this, there are others with the same problem and they will want to contribute”

Therefore, shared causal beliefs have a co-ordinating effect on the development process it makes sure that patches are within the frame of reference for the project. Patches using coding style or principles different from the ones applied in the project will have difficulties being accepted in the project.

3.1.3 Shared notions of validity

As mentioned above programming problems are usually solved as contributors suggest different solutions. The choice between these patches is eased because the contributors share notions of validity. There shall be little doubt that the first contributor would rather like his patch applied than that of the second contributor. Effort has gone into develop the software that turns into a patch, which are submitted to the project. Clearly there is little credit gained from making the second best solution. Despite these very personal incentives to defend ones patch the decision rarely ventures into personal crusades, although this has been seen.

Validity tests are an important part of open source software development. Different solutions are evaluated using two criteria: 1) Performance and 2) Beauty.

The raw performance tests of different designs are easy to understand. A benchmark is developed to simulate the intended usage of the software. The benchmark is then tested on the proposed patches. The scores of the different designs are compared and a choice is made

based on the best performer. Sometimes the actual benchmark tests become the centre of debate, as it simulates a particular use of the program. Whether or not the simulation is accurate, it leaves ample room for personal interpretation. Despite the potential trying to create a synthetic benchmark, it proves the performance of a person's particular patch. The discussion resides on a level where the technical and logical argument is considered most valuable.

Beauty is an elusive term used by all the developers, I have had the pleasure of interviewing. When discussing how the developers decided for themselves what is a good solution to a problem, the answer was "It looks good", "It is beautiful", and "It is nice". It was not possible to get a precise definition of Beauty, however, among the interviewed developers there was an implicit understanding of what was a right solution. Readers of Kernel Traffic will remember several occasions where Linus Torvalds has rejected a patch that solved the problem, but the solution 'looked' wrong. The right solution is simple and easy to understand, it is logical and solves the problem with a minimum of workarounds. Often a Beautiful solution is the product of insight and a clear understanding of the problem to be solved. Understanding the complexity of the software and making the right assumptions in the software helps reduce the amount of code otherwise required to check these assumptions. There is also a maintenance perspective in Beautiful solutions. If they are logical and simple, they will also be easy to expand and develop further. Whereas a solution that requires many workarounds is difficult to understand for other than the person who made the solution. Such solutions are therefor not suited for further development and commonly referred to as an 'ugly hack'. Most projects, particularly the kernel, are very keen on implementing good solutions and lot let badly written code pollute the long-term viability of the project.

The actual process of choosing between alternatives, be it performance or beauty criteria, can be viewed as a power struggle between stakeholders. The different stakeholders try to influence each other's opinion. In epistemic communities, the power exerted has to come to terms with the peculiar quality of power in the specific epistemic community (Holzner and Marx, 1979, p. 109). An epistemic community has a primary commitment to a certain type of knowledge, in open source software this is the knowledge of programming in general and the experience gained in the particular project. This impacts the way power is exerted in the community, which again require a certain legitimisation. The primary commitment to a certain kind of knowledge defines not only the nature of work in the community but also the terms of its legitimisation (Holzner and Marx, 1979, p. 109). Discussion must use the terms and arguments that the community views as valid. This, too, depends on the history of the particular project; however, logical argument backed up with data to prove a point is generally perceived legitimate arguments. Usually there is no point in shouting and slamming the door – no one will listen. There are of course noteworthy exceptions. Linus Torvalds, who created the Linux operating system, sometimes makes decisions and presents arguments, which are not open to the usual discussion of OSS projects:

> "a 32-bit (or 64-bit) dev_t does NOT make it any easier to manage permissions or anything like that anyway. Look at the current mess /dev is. Imagine it an order of magnitude worse.
>
> Big device numbers are _not_ a solution. I will accept a 32-bit one, but no more, and I will _not_ accept a "manage by hand" approach any more. The time has long since come to say "No". Which I've done. If you can't make it manage the thing automatically with a script, you won't get a hardcoded major device number just because you're lazy.
>
> End of discussion." (Torvalds, 2001)

On several occasions Linus Torvalds has used harsh language and behaved less than polite to the people contributing to the kernel development. Due to his very high status in the community, he has got away with his temper and bad behaviour. In this particular incident, Linus Torvalds exertion of power generated a negative counter reaction. A few top

developers publicly discussed whether to continue supporting Linus Torvalds and instead support Alan Cox, who maintains the parallel ac-series of the Linux kernel.

In the above quote Linus Torvalds' arguments were not perceived as legitimate by the community and this promoted a counter reaction. This demonstrates the OSS community as an epistemic community where even the charismatic leader must acknowledge the means of excerting power, which are perceived as legitimate.

3.1.4 Common Policy Enterprise

Participators in OSS development also have the common policy enterprise. This could be stated as a wish to replace all commercially closed source software with open source alternatives. The Free Software Foundation, which authored the GPL open source license, has made this an explicit goal.

Sometimes the problem is defined by the common policy enterprise and OSS projects are formed to create an open source alternative. Projects like GNUcash and Appogen are inspired by the commercial software Quicken and support the Quicken file format.

The common policy enterprise serves both as a motivating factor and as shared understanding among participators in the open source software community. In this perspective the community defines itself as a counter part to closed source commercial software. The open source software community is small compared to users of Microsoft windows. There is an immediate understanding among users in the OSS community that crosses established social boundaries. As an attendant at Linux user group meetings and conferences, I find that there is an interesting mix of people, which would most likely not be possible under other circumstances.

The common policy enterprise lowers the personal barriers to engage in discussion and defines common themes of discussion when entering a dialog with a stranger. Examples of

common themes are: Microsoft bashing, promoting ones favourite free OSS at the expense of a proprietary alternative, technical discussion regarding personal use and choice of software, discussing how to promote open source software.

3.2 Shortcomings of the Epistemic Communities Approach

The understanding of open source software development using an epistemic communities approach appears useful. It is possible to understand the motivational issues of participation in OSS development. Contributors are motivated by a personal need for software. When contributors have entered the community, another motivational factor influences and contributors are met with reward in terms of reputation and recognition from ones peers. There shall be little doubt that the reputation game is a force, which has to be recognised when trying to understand OSS development. The effect of the reputation game varies according to personality; some developers have expressed a kind of addiction towards maintaining their reputation. Interview with FreeBSD[5] developer Poul-Henning Kamp (2000) revealed that developers once considered great and important contributors were having problems when scaling down their development effort. Those developers had grave difficulties coming to terms with their lower influence. They were still following the development of the project and monitoring changes made to 'their' part of the code. Changes they did not like were criticised in a way that was considered grumpy and inappropriate for retired developers. This has led to cases where the community has shunned the developers (Kamp, 2000).

We are also able to understand the actual development process using the shared understanding of normative, principled, causal beliefs and notions of validity, and the

[5] FreeBSD is a free operating system, derived from the Berkeley Software Distributions, URL: www.freebsd.org

common policy enterprise. These factors ease collaboration in the project by minimising the need for co-ordination and communication. The factor makes sure that problems are perceived in roughly the same way, and also solutions and added features may be implemented with little or no interaction with the community. It must be remembered that most, if not all, development takes place in the private sphere of the individual developer. This illustrates a community where the participators share a mindset – a way of thinking and understanding the problem.

This is indeed needed if this kind of distributed development is expected to produce any results. The limited communication's bandwidth requires that participators are able to communicate using as few words as possible. Unlike a company, where employees are physically placed in the same building and have the opportunity to talk and discuss mattes directly with each other, OSS development only has the means of communications provided by the internet. Most of these forms of communication are time shifting forms of communication, chat and voice/video conferencing being noteworthy exceptions. Real-time communications are, however, not suitable for OSS development simply because people are not working at the same time. Some developers are living in Denmark and working on projects with persons from Mexico and the United States, and difference of time alone makes it difficult for to meet using real time communications. If one thinks of collaboration as a continuing exchange of ideas and contributions to the project, the level of collaboration in OSS development is actually quite small. OSS development is very individualistic, and collaboration is better described as contributions to the project.

Contributors rely on a common frame of reference (the four characteristics) as a mechanism for co-ordinating their efforts when venturing into OSS development. This is not a conscious reliance, but a consequence of being part of community and solving a common programming problem. It is not possible, given the limited means of communications, to

socialise people into the community and create that shared frame of reference, which has a significantly different mindset. The effort to accomplish this task would be far greater than the benefit. OSS contributors are motivated by need and interest in solving the problem, and far less by socialising.

And here we touch the heart of problem. Persons wanting to contribute can not be expected to share the same mindset as the community. How should we then understand the process of socialisation in OSS development?

4 Situated Learning and Legitimate Peripheral Participation

The process of entering and becoming a member of an epistemic community is, as noted, not readily explained in the epistemic communities approach. This is an important process as it is the key to understanding the population and reproduction of the epistemic community. This understanding may also be used to ease the process of becoming a member. Many 'wanabe' developers and newcomers to OSS development have difficulties understanding how to contribute to the development. Often project websites contain instructions of how to contribute, but these instructions miss some of the finer points on proper conduct and behaviour. It seems that these finer points are important and the distinguishing point between becoming a part of the community and staying on the outside.

I shall not enter into at discussion of what are the 'finer points' in participating in OSS development. They will wary depending on the particular epistemic community.I shall, however, propose a framework for a preliminary analysis and understanding of OSS development epistemic communities.

I propose *situated learning* and *legitimate peripheral participation* (LPP) as the analytical tools for understanding of the process of socialisation into OSS development. This

requires the socialisation process to be understood as a learning process, where people wanting to participate become participators through a learning process.

Theories of learning often view learning as a transmission of knowledge in some abstract form from one person to another. This perception of learning excludes all the details, which links knowledge to a specific situation in which knowledge is useful and makes sense. This kind of transfer models have been severely attacked, and in particular Lave and Wenger (1991) with their concept of situated learning have shown another way of understanding learning.

Situated learning ties learning to the actual situation in which the knowledge should be used. It is impossible to separate knowledge from context, which also include the social situation in which the knowledge is used. The learners construct their understanding from a wide range of parameters including ambient social and physical conditions, and the history and social relations of the people involved (Brown & Duguid, 1995, p. 69).

The interesting point when relating LPP to OSS developing epistemic communities is that learning is a process of becoming a practitioner. Learners do not receive abstract knowledge de-coupled from any practical use. Learners are becoming practitioners in a community by being part of that very community and acting as apprentices. This requires that apprentices be viewed as legitimate peripheral practitioners by practitioners of the community. Practitioners must allow learners to participate at a learner's level. By observing and participating, the learner acquires knowledge, skill and understanding of the social sphere of which he is a part.

4.1 Open Source Software and Legitimate Peripheral Participation

When using LPP and situated learning for understanding OSS development, I define two different groups' *insiders* and *learners*. Insiders are members of the community who are

active in the community and participate in the activities of the community. Insiders are practitioners who function in the community. Learners are persons who wish to become part of the community and participate in the project. In short, learners want to become insiders.

The process of becoming an insider requires the learner to learn the ways of the community, how they act, interact, etc. This is not something that can be taught as noted in an earlier example, and descriptions of how to enter the community seem to miss the finer point. The learning process has an individual bias depending on the learner, which is affected and influenced by the learners' relations in the community. Becoming an insider can only be learned, and this is a situated process where the learner participates in the actual and practical life of the community.

By participating, the learner observes the practice of the other members of the community. What is more important is that when the learner participates, he tests the different strategies or ways of interacting with the other members of the community. In the course of this process, the learner begins to form his own impression of the inner workings of the community.

Completion of this process and transformation of the learner into an insider requires that the insider *must* allow the learner legitimate peripheral participation. Insiders must acknowledge the fact that learners are just learner'. Learners cannot be expected to perform at the level of the insiders, and insiders must respect their effort. It must be legitimate for learners to participate in the practice of the community. Learners are bound to a peripheral position when participating. Learners do not have the skill of the insiders, and must accept a position where they are allowed to participate at their own level and observe.

The learning process in open source software development takes place in two spheres 1) The private and 2) The collective sphere.

1) The Private Sphere

This is the development situation in the home of the individual developer. In this situation the developer tests and explores different solutions to problems, which he wants to solve. The development is a recursive process where different principles are implemented in order to truly understand the problem and make the 'right' solution.

2) The Collective Sphere

This is where the developer interacts with the epistemic community using mailing lists, chat, news groups, and other available means of communications. Questions, answers, and discussion are communicated back and forth between the community and the members.

The two spheres are by no means separate but exist side by side. Often developers have a keen eye on what is happening on different mailing lists and in chat rooms when they are working. Developers Jens Axboe[6] (2000) and Kenneth Rohde Christensen[7] (2000) explained how they always, like many of their developer friends, had their email client and chat client running while working. The two developers noted that the community feeling was quite strong when they logged on to a chat forum and were greeted by friends.

The learning situations of the two spheres are different but both require understanding of the practice of the community. The private sphere requires understanding of the practice of coding - the coding style – in the project. Most projects have a special way of writing codes, and this entails the use of indentation, proper use of functions etc. Adhering to the project coding style ensures readability of the code by other members of the community. This is mostly about how to code the project software. Much of the learning in the private sphere is formalistic and can thus be taught without having to engage in the community.

---

[6] Jens Axboe is CD-ROM maintainer and employed by SuSE..

[7] Christiansen, Kenneth Rohde is a GNOME developer and language translator.

Becoming a practitioner in the collective sphere, on the other hand, cannot be taught. Situated learning and legitimate peripheral participation are important in this sphere. Learners engage in the practice of the collective sphere, and by doing so they become practitioners, who should be allowed legitimate participation in the project.

Becoming a practitioner and going from learner to insider is the process by which OSS developing epistemic communities recruit new members. It is a process, which requires the learner to actively take part in the project and contribute. It is also a process where the learner must expect that becoming an insider happens gradually. The transition from learner to insider is subtle, and the learner will often be the last to call himself an insider. In the process, the learner adopts the language, the coding style etc. of the community. The learner has then become an integral part of the community.

5 Conclusion

This paper has tried to explain the development process of open source software and has attempted to apply theory from other disciplines. Epistemic communities, situated learning, and legitimate peripheral participation have been used for understanding of the development process of open source software. The combination of theories was necessary for understanding of both the development process and entry into the epistemic community of open source software development. In itself the exercise of testing a theoretical body on new empirical problems is a worthy effort. Nevertheless, the question remains: Has this approach brought forth a new insight to understanding open source software?

I believe that this approach has actually produced new insights. In the frame work of epistemic communities it is possible to understand how open source software can be developed under difficult circumstances,, and when simple economic incentives are not present.

It has been shown that the dynamics of epistemic communities offer a fair explanation of the organisation and development direction of the projects. However, the dynamics of epistemic communities do not provide insight into what motivates people to participate in OSS development. To this end, personal need for a particular piece of software has been introduced.

Haas' four characteristics of 1) Shared normative and principled beliefs, 2) Shared causal beleifs, 3) Shared notions of validity and 4) A common policy enterprise is useful for understanding OSS development. They provide adequate explanation as to how participators in OSS development are able to co-ordinate their actions in subtle and rarely explicit ways. There is a mutual understanding, a common frame of reference of what to develop and how to do it.

This common frame of reference does not occur instantaneously, and a process of socialisation into the community precedes the state of mutual understanding. The process should be understood as a learning process, where newcomers to the community are perceived as learners in the process of becoming insiders. For the process to be successful, insiders must allow learners to be legitimate peripheral participators. It is important that learners are allowed to be practitioners in the community at their own level. Learners must grow into the community and through participation form their own understanding of the community. With interaction, this understanding is reflected in the other practitioners and a mutual understanding is formed.

As this process gradually completes, the learner becomes an insider and a practitioner in the community. At this point, the learner has become part of the epistemic community.

6 References

Axboe, Jens is CD-ROM maintainer employed by SuSE, Interview conducted the 11th of September 2000.

Bessen, Jim, 2001, “Open Source Software: Free Provision of a Complex Public Good”, URL: http://www.researchoninnovation.org/opensrc.pdf.

Brown, J. S. & Paul Duguid, 1995 “Organisational Learning and Communities of Pratice” in Eds. Cohen, M. D. & L.S. Sproul “Organvisational Learning”, SAGE Publication, Inc.

Christiansen, Kenneth Rohde is a GNOME developer and language translator. Interview conducted the 21st of September 2000.

Edwards, Kasper, 2000a “When Beggars Become Choosers”, First Monday, volume 5, number 10 (October 2000), URL: http://firstmonday.org/issues/issue5_10/edwards/index.html

Edwards, Kasper 2000b, working paper, “The history of Unix development” URL: http://www.its.dtu.dk/ansat/ke/emp_work.pdf

Edwards, Kasper, 2001, “Towards a Theory for understanding the Open Software Phenomenon”, paper written as part of a Ph.D. knowledge test. URL: http://www.its.dtu.dk/ansat/ke/towards.pdf

Holzner, B. and J. Marx, 1979, "Knowledge affiliation: the Knowledge system in society" Allyn and Bacon, Boston, MA.

Haas, P. M., 1992. "Introduction: Epistemic Communities and international policy coordination", International Organization 46, 1.

Kamp, Poul Henning, interview conducted 27th of September 2000.

Kernel Traffic, 2001, URL: http://kt.zork.net/kernel-traffic/kt20010521_119.html#stats

Koch, Stefan & Geog Schnider, 2000, “Results from Software Engineering Research into Open Source Development Projects Using Public Data”,URL: http://exai3.wu-wien.ac.at/~koch/forschung/sw-eng/wp22.pdf

Lave, J. & Wenger, E., 1991, "Sitauted Learning - Legitimate periphal participation", Cambridge Uni Press.

Learner, Josh and Tirole, Jean, March 2000, “The Simple Economics of Open Source”, Working Paper 7600, National Bureau of Economic research, 1050 Massachusetts Avenue, Cambridge, MA 02138

Levy, Steven, 1984, “Hackers: Heroes of the Computer Revolution”, Anchor Press/Doubleday, Garden City, NY.

Raymond, Eric S. 1999a, “Homesteading the Noosphere”, in The Cathedral & the Bazaar, O’Reilly and Associates.

Raymond, Eric S. 1999a, “The Cathedral and Bazaar”, in The Cathedral & the Bazaar, O’Reilly and Associates.

Salus, Peter 1994 “A Quarter Century of Unix”, Addison-Wesley Publishing, Inc.

Temporary Committee on the ECHELON Interception System, 2001, “Document on the existence of a global system for intercepting private and commercial communications (ECHELON interception system)” URL: http://www.europarl.eu.int/tempcom/echelon/pdf/prechelon_en.pdf

The Linux Study, URL: http://www.psychologie.uni-kiel.de/linux-study/index.html

Torvalds, Linus, 2001, Kernel Traffic, URL: http://kt.zork.net/kernel-traffic/kt20010528_120.html#5

Tuomi, Ilkka, 2000, “Internet Innovation and Open Source: Actors in the Network”, URL: http://opensource.mit.edu/papers/Ilkka%20Tuomi%20-%20Actors%20in%20the%20Network.pdf